To appear in ACM TOG 30(6).

Coherent Parallel Hashing

Ismael García[1,2] Sylvain Lefebvre[2] Samuel Hornus[2] Anass Lasram[2]

[1]GGG / Universitat de Girona [2]ALICE / INRIA

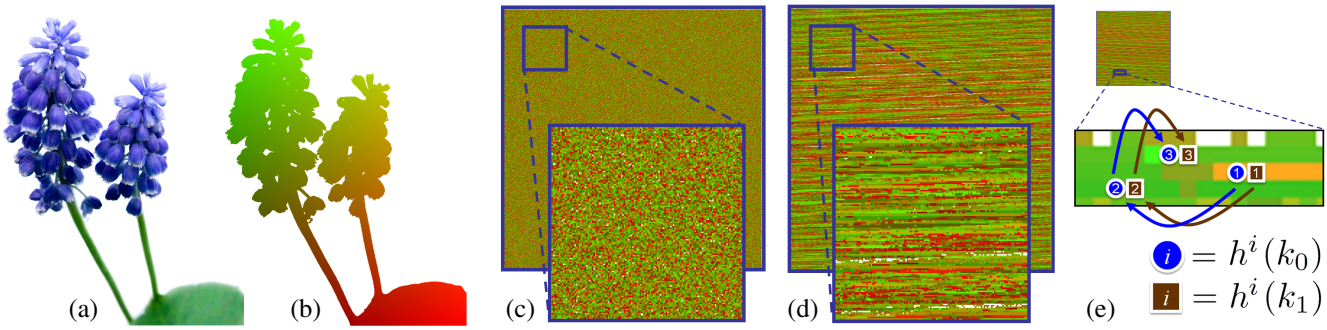


Figure 1: (a) *The flower image is* 3820×3820 *image (*14.5 *million pixels) and contains* 3.7 *million non–white pixels. The coordinates of these pixels are shown as colors in* (b)*. We store the image in a hash table under a* 0.99 *load factor: the hash table contains only* 3.73 *million entries. These are used as keys for hashing.* (c) *The table obtained with a typical randomizing hash function: Keys are randomly spread and all coherence is lost.* (d) *Our spatially coherent hash table, built in parallel on the GPU. The table is built in* 15 *ms on a GeForce GTX 480, and the image is reconstructed from the hash in* 3.5 *ms. The visible structures are due to preserved coherence. This translates to faster access as neighboring threads perform similar operations and access nearby memory.* (e) *Neighboring keys are kept together during probing, thereby improving the coherence of memory accesses of neighboring threads.*

Abstract

Recent spatial hashing schemes hash millions of keys in parallel, compacting sparse spatial data in small hash tables while still allowing for fast access from the GPU. Unfortunately, available schemes suffer from two drawbacks: Multiple runs of the construction process are often required before success, and the random nature of the hash functions decreases access performance.

We introduce a new parallel hashing scheme which reaches high load factor with a very low failure rate. In addition our scheme has the unique advantage to exploit coherence in the data and the access patterns for faster performance. Compared to existing approaches, it exhibits much greater locality of memory accesses and consistent execution paths within groups of threads. This is especially well suited to Computer Graphics applications, where spatial coherence is common. In absence of coherence our scheme performs similarly to previous methods, but does not suffer from construction failures.

Our scheme is based on the Robin Hood scheme modified to quickly abort queries of keys that are not in the table, and to preserve coherence. We demonstrate our scheme on a variety of data sets. We analyze construction and access performance, as well as cache and threads behavior.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types;

Keywords: spatial, parallel, coherent, hashing, sparse data

1 Introduction

Sparse spatial data is very common in Computer Graphics and finding the good tradeoff between access performance and efficient storage is an ongoing challenge. Spatial hashing has proven useful in these situations, enabling data to be tightly packed while still allowing fast random access. It has been successfully applied for texturing, rendering, collision detection and animation.

Using a spatial hash, data is stored in a single array—the *hash table*—addressed through a *hash function*. The hash function computes the data location in the hash table from the query coordinates, or *keys*. There have been several developments lately, improving query and construction times, and in particular enabling fast parallel construction on GPUs.

The first spatial hashing schemes focused essentially on reaching good load factors while having a constant time and simple access to the data from the GPU. Lefebvre and Hoppe [2006] proposed a static hash construction enabling access to the data with as little as two memory accesses and one addition. However, to achieve this result the hash has to be *perfect*: All keys corresponding to defined data should map to different locations in the hash table. In other words, there are no *collisions*. Building such a constrained hash function requires an off–line construction process, limiting this approach to static cases.

Alcantara *et al.* [2009; 2011] propose less constrained hashing schemes that achieve fast, parallel construction on the GPU. These schemes may produce collisions. However, querying a key never requires more than four independent memory accesses. The particular hash mechanism they use is known as *Cuckoo hashing*. We detail it in Section 2.

Both approaches achieve constant query time with a fixed number of instructions. Unfortunately, these constraints imply that construction is difficult and the process may fail, requiring several restarts especially at high load factors. In this paper, we relax the constraint of accessing data with a fixed number of instructions. Instead, we implement queries with few memory fetches *on average*. The increased flexibility in the access enables a more robust construction process. However, a small average does not guarantee good performance as empty keys are generally detected only after

trying the *maximum* number of accesses required to find a defined key. We propose a mechanism to quickly reject empty keys, thereby significantly reducing their negative impact.

In addition, we tailor our scheme to exploit the spatial coherence of rendering algorithms. In existing schemes, neighboring keys are often mapped to distant locations in the hash table. This is an issue since graphics hardware is designed to benefit from spatial coherence: Threads are organized in a grid and best access performance is achieved when nearby threads access nearby memory locations. Lefebvre and Hoppe [2006] were aware of this and proposed a construction process preserving some degree of coherence, however with only limited positive impact on access performance. Linial and Sasson analyzed a *non-expansive* hashing scheme to bring similar keys close to each other in the hash table [1996]. That scheme, however, necessitates too much space to be practical in graphics applications. As we shall see, the improved robustness of our scheme lets us design hash functions improving memory coherence during queries, while still affording high load factors.

Contributions Our main contributions are: 1) A parallel hashing approach reaching high load factor with a low failure rate. It relies upon a coherent hash function exploiting coherence in memory accesses, when available. This leads to increased locality in memory accesses and increased coherence in the execution paths within a thread group accessing the data. 2) An improved query scheme using a few bits of additional information per key to efficiently find defined keys, and perform early rejection of empty keys.

We introduce a complete parallel GPU implementation and analyze its behavior in details.

Notations and definitions Keys are taken in a universe U of size $|U|$. We note $D \subset U$ the set of *defined keys*, that is the keys from U which should be stored in the hash table. Keys which do not belong to D are called *empty keys*. Throughout the paper we consider the *load factor* d. It corresponds to the ratio of the *number* of defined keys to the number of keys that the hash table can hold. A hash table with load factor $d = 1$ is a minimal hash, that is a hash with no wasted space. It is worth considering another type of "load factor", which we call the *key-density*; it is defined as the ratio of the number of bits used to store the keys to the number of bits in the hash table (not counting data bits).

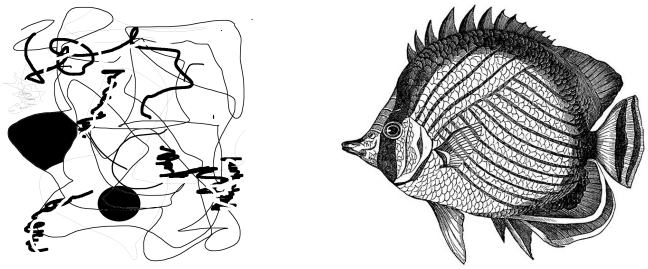
Each defined key may be associated with some additional data. In our setting, all defined keys are associated with data fields having a same, fixed size (e.g. an RGB color triple, or a boolean value). The input is thus given as a set of key-data pairs.

An application is said to perform a *constrained access* to the hash if the only queried keys are in the set D. The scheme of [Lefebvre and Hoppe 2006] is especially efficient in this situation, as keys do not need to be stored. In this paper however we are mostly concerned with the *unconstrained* access scenario, where empty keys may be queried and must be detected as such.

By *coherence* we refer to the locality of the parallel memory accesses performed within a thread group. In Section 4.2 we compare query performance for the 2D case using different access patterns: Linear row major, Morton and the bit-reversal permutation. The first two offer a strong locality—neighboring threads access neighboring data—while the third has poor locality.

2 Background on parallel hashing

Alcantara *et al.* [2009] introduced the first algorithm enabling fast, parallel hash table construction on the GPU. Millions of keys are efficiently hashed in milliseconds, outperforming previous schemes by several orders of magnitude. Since this approach is the closest to our work, we describe it in more details.



test1: $2.2M/4096^2$ fish: $20.5M/8192^2$

Figure 2: *Two of the datasets used to test our hashing algorithm. They give a good spread of behaviors between randomness and structure. The number of defined keys (black pixels) and the size of the image is indicated below each.*

A Cuckoo hash [Pagh and Rodler 2004] maintains two or more different tables of the same size, each accessed through a different hash function—Alcantara *et al.* [2009] use three tables. Keys are inserted in the first table, evicting already inserted keys in case of collision. Evicted keys are in turn inserted in the second table, then from the second to the third, and from the third back to the first. The process loops around until all keys are inserted or until the number of iterations reaches a maximum—which triggers a construction failure. Upon failure the process is restarted with different hash functions. Unfortunately, given a number of tables the failure rate abruptly increases above a limit load factor. Using more tables increases this limit. However, using too many tables becomes wasteful at lower load factors. In contrast, our scheme automatically adapts to these various situations.

The parallel construction algorithm builds a Cuckoo hash in shared memory—a small but very fast memory. It starts by randomly distributing the keys in equally sized buckets using a first level hash [Botelho and Ziviani 2007]. Any bucket overflow triggers a construction failure. This limits the maximum possible load factor to 0.7: higher load factors give a too large failure rate for this key distribution phase. Next, all buckets are hashed independently in parallel with Cuckoo hashing. In recent work, Alcantara *et al.* [2011] build a Cuckoo hash in a single pass. The single pass approach is made possible by latest hardware capabilities (efficient atomic operations on NVidia Fermi devices). Their new hashing scheme also reaches higher load factors thanks to the use of four hash functions. Both schemes further introduce handling of multi-value hashing and duplicate keys in the input.

The Cuckoo scheme behaves very well in practice, and the guarantee of a constant number of memory accesses to query a key is well adapted to GPUs. Its main drawback stems from an increasing failure rate at high load factors requiring to manually select more hash functions, and the loss of coherence due to randomization. Another less obvious issue is that while a fixed number of lookups are required, the average number of lookups is often higher than that of our scheme as keys tend to be uniformly distributed in all the tables, even at lower load factors. We compare our work to parallel Cuckoo hashing in Section 4.

3 Parallel coherent hashing

Our hash is designed to reach high load factors at low failure rate, and to provide fast queries regardless of the load factor.

The key insight is to exploit dynamic branching to release the constraints on the construction process, and to exploit coherence in the access patterns when available.

Our algorithm builds a unique, large hash table in one pass. Parallelism is obtained by launching many thread groups simultaneously. Each thread is responsible for inserting exactly one key.

3.1 Main algorithm

Our hash is at heart an open addressing scheme [Peterson 1957]: Each input key k is associated with a sequence of probing locations $h^1(k), h^2(k), \dots$ in the hash table. Ideally, this *probe sequence* should enumerate all locations in finite time. For now, we assume that the sequence is given. We introduce our coherent probe sequence later.

In order to add a key, the insertion algorithm iterates along the probe sequence until an empty location is found. The key is then inserted. We call the number of steps required for successful insertion the *age* of a key. If the hash table can fit all the defined keys, then the process is guaranteed to succeed as long as the sequence $h^i(k)$ enumerates all locations. Therefore the algorithm is quite robust to changes in the hash function. Querying a key proceeds similarly to construction, by walking along the probe sequence until the key is found in the hash table.

However, open addressing suffers from a severe drawback for our purpose: The age of the keys is very low on average but typically has a large maximum. This maximum age, noted M, is crucial: When querying an empty key, its absence from the hash table can only be verified by walking along the sequence of keys at least M steps. Since the data set is sparse, a large number of queries to empty keys is expected in many applications. The overall performance can dramatically suffer. Note that hitting an empty location during a query before reaching M steps is unlikely, especially under high load factors.

We next discuss how to efficiently reduce the maximum age and reject empty keys.

Reducing the maximum age The maximum age issue has already been identified and studied in previous work. A very effective solution to it is known as Robin Hood hashing [Celis 1986], which is based on open addressing.

The idea is to store the age of the keys in the hash table during its construction. This additional data is discarded afterwards. Consider the case of inserting a key k_{new} at a location $h^i(k_{\text{new}})$ already occupied by a key k_{prev}. The age a of k_{prev} is compared to i. If the key being inserted is older ($i > a$), then k_{prev} is evicted. The current key is inserted at $h^i(k_{\text{new}})$ and k_{prev} is recycled into the set of keys to be inserted. Intuitively, the keys which are difficult to insert push away the keys which have been easier to insert. This has a drastic effect on the histogram of key ages, and in particular on the maximum age as illustrated Figure 3.

There are two particularly important theoretical facts about Robin Hood hashing making it especially well suited to our purpose [Celis 1986]: The expected maximum age in a *full* table of size n is $\Theta(\log n)$ and Devroye *et al.* [2004] improve the bound to $\Theta(\log_2 \log n)$ on non full tables. Furthermore, the expected query time complexity can be made constant if starting the accesses at the average age. Note that these facts are derived assuming uniform random sparse data, but our experiments show that they hold in practice on our datasets.

In our current implementation we do not start the accesses at the average age: For simplicity we always start from age one, searching for the key until the maximum age for the sequence is reached.

Figure 3: *Hashing 2^{20} defined keys randomly distributed in a universe of 2^{24} keys into a hash table of 1.3 million keys (the load factor is 0.8). Histogram of insertion ages for open addressing (top) and Robin Hood (bottom). Gray bars correspond to very few items but are non–zero. The maximum age goes down from 46 to only 5.*

Empty key rejection While Robin Hood hashing strongly reduces the maximum age M, it remains quite large compared to the few memory accesses of Cuckoo hashing. This is especially important in applications querying many empty keys, as they always require M steps along the sequence. We therefore introduce a new mechanism to accelerate the rejection of empty keys.

We note that most keys have a very small age, with only a few outliers ever reaching M. Therefore, in most cases a much smaller value than M would suffice to detect empty keys. To benefit from this we store in each entry of the hash table the maximum age of all the keys mapping *first* to this location. More precisely, let $H[p]$ be the key stored at location p in the hash table H, let $\text{MAT}[p]$ be the maximum age starting from p, then we have:

$$\text{MAT}[p] = \max_{\{k \in D | h^1(k) = p\}} \Big(i \text{ st. } H[h^i(k)] = k \Big) \qquad (1)$$

When querying a key k we iterate at most $\text{MAT}[h^1(k)]$ times along the probe sequence. This guarantees that defined keys are found, and affords for fast detection of empty keys.

Encoding the maximum age Storing the maximum age table MAT requires additional memory. Fortunately, the maximum age values are small and only require a few bits. In all our tests, the maximum age was below 16. Thus, we reserve 4 bits in each hash table cell to store the maximum age. The latter can optionally be quantized to either accommodate larger values or reduce even further the number of bits used to 3 or less.

Let us assume each key is stored on k bits and their age is stored on a bits. When the user targets a data structure p times larger than the defined keys, we allocate $pk|D|$ bits of memory. In fact, due to the additional storage of the maximum age, this corresponds to a hash table storing $\frac{pk|D|}{(k+a)}$ keys. Thus, the keys will be hashed at load factor $\frac{(k+a)}{pk}$. For example, in a typical situation, we have $k = 28$ bits, $a = 4$ bits. Then, targeting a storage of $1.2 \times k \times |D|$ bits—that is 1.2 times the size of the defined keys, or a 0.83 key-density—requires hashing at 0.95 load factor, while targeting a 0.7 key-density requires a 0.82 load factor. With 64 bit cells and 60 bit keys, these load factors become 0.89 and 0.76 respectively.

The hashing scheme of Alcantara *et al.* [2011] requires no additional information apart from the key, in which case the load factor and the key-density are equal.

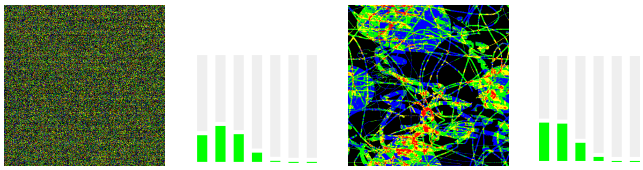


Figure 4: *Maximum age table for* test1 *hashed at* 0.8 *load factor. Color code: 0, black; 1, blue; 2, green; 3, yellow; greater, red.* Left: *Random probe sequence.* Right: *Coherent probe sequence. Key age histograms are comparable, but note how coherence is maintained in the map on the right. We can expect more efficient dynamic branching when branching with respect to this map.*

Exploiting coherence In many computer graphics applications, data is queried in a coherent manner: Either spatial coherence within a frame, or temporal coherence due to limited motion between frames. We design a new probe sequence tailored to exploit coherence in the queries.

Note that we seek coherence of memory accesses *between neighboring threads*. This is quite different from the typical CPU notion of cache coherence where one seeks to access nearby memory locations *in sequence*. On the GPU, groups of threads access memory *simultaneously*, and it is important to group the accesses in nearby locations. This is also known as *coalesced* accesses. Similarly, the hardware performs faster when groups of threads take similar branching decisions.

Our objective is to design a different probe sequence for the keys, which favors coherence. Our new probe sequence h_{coh} preserves coherence by making neighboring keys test neighboring locations—while still ensuring that the *successive* locations of a same key are uncorrelated and perform a random walk. It corresponds to random translations of the *entire* hash table at each step i. That is, for a key k at step i in a hash table of size $|H|$:

$$h^i_{\text{coh}}(k) = k + o_i \mod |H|$$

where o_i is a sequence of offsets, independant of k. We typically set $o_0 = 0$ and use large random numbers as offsets.

An important property of this probe sequence is that neighboring keys remain neighbors at each step, as illustrated Figure 1(e). Therefore, if two neighboring threads attempt to find neighboring keys, they will both always access neighboring memory locations until one terminates. Note that these keys do not have to be present in the hash table for coherence to happen at the thread level during queries. In fact, access coherence during queries is orthogonal to the distribution of the defined keys. Sparse random data encoded with our hash will still benefit from being queried in a coherent manner.

The map MAT encoding the maximum ages also benefits from a coherent hash function: It exhibits a much stronger spatial coherence. This implies that neighboring threads will perform similar data–dependent loops, reducing divergence. In Figure 4, for illustration purposes we hash the image test1 shown in Figure 2 in a 2D hash table – we extend the hash to 2D by applying the same computations independently to each dimension. We display the maximum age table MAT for a random and a coherent probe sequence. The second image (coherent) exhibits structure and coherence, and thus affords for more efficient dynamic branching than the first.

Our coherent probe sequence outperforms the random one when coherence is present. It strongly reduces cache misses, thread branch divergence and results in significantly faster queries. We measure these effects on various data sets in Section 4.

3.2 Implementation

Our CUDA implementation runs on a graphics processor, where multiple threads are organized in groups and execute the program in parallel. We always build 1D hash tables. 2D and 3D data is linearized, as discussed in Section 4.2.

Building the hash table in a single pass requires global atomic operations to safely manipulate memory. Our eviction strategy requires comparing the age of the key to be inserted to the age of the key already in the hash table. These operations—compare ages and store key if greater—have to be performed atomically, or the table will quickly get corrupted by concurrent accesses.

We encode the age, the key and its data in a single word (either 32 or 64 bits) with the age on 4 bits. We never observed an age above 15 in all our tests, but reaching this value would trigger a construction failure. The 4 bits of the age are stored in the most significant bits. This is important since on 32 bits, for two words $w = a \times 2^{28} + k$ and $w' = a' \times 2^{28} + k'$ we have $a > a' \Rightarrow w > w'$. Thanks to this property we can test and store in a single atomic max instruction.

The insertion algorithm is detailed next. For clarity we ignore the data fields. Please note that this is a pseudo code. The actual implementation differs slightly. In particular, the `atomicMax` operation is not available on 64 bits words on current hardware (however, it is natively supported on 32 bit words). We thus emulate it using `atomicCAS`, as suggested by the CUDA programming guide. This incurs a performance penalty in the construction process: Full hardware support of 64 bits `atomicMax` will further improve performance.

The input defined keys are in the array `D`. The outputs are the hash table `H` and the max age table `MAT`. The type `uint` represents unsigned integers. The algorithm performs the following operations:

```
1  kernel(const uint *D, uint *H, uint *MAT) {
2    uint key          = D[ global_thread_id ];
3    uint age          = 1;
4    while( age < MAX_AGE ) {
5      uint h_k_i    = hash( key , age );
6      uint age_key = PACK( age , key );
7      uint prev     = atomicMax( & H[ h_k_i ] , age_key );
8      if ( age_key > prev ) {
9        uint h_k_1 = hash( key , 1 );
10       atomicMax( & MAT[ h_k_1 ] , age );
11       if ( AGE( prev ) > 0 ) {
12         key      = GET_KEY( prev );
13         age      = GET_AGE( prev );
14       } else {
15         return;
16       }
17     } else {
18       age++;
19     }
20   }
21 }
```

init The hash table `H` and the max age table `MAT` are allocated and initialized to zero.

2-3 The thread reads the key from the input located at index `global_thread_id`, which is unique for each thread. The current key is read in `key`, and `age` is the current insertion age.

4 The thread executes until the key has been inserted in an empty cell of the hash table or the maximum number of iterations for the current key has been reached.

5 The hash function is applied to `key` at `age`.

6-7 The key and its age are packed into a word `age_key` (32 or 64 bits). An `atomicMax` is used for the eviction mechanism. This instruction compares the current value in memory to `age_key` and replaces it if greater. The previous value is always returned.

8 Tests whether an eviction occurred, by comparing the value returned by `atomicMax` to `age_key`.

9-10 An eviction occurred and the current key has been inserted. These two lines update the max age table `MAT`. The first hash position of the current key is computed, and an `atomicMax` updates the max age table.

11-13 The age at the insertion location determines whether it was an empty slot or a previously inserted key. An age above zero implies that a key was evicted. The evicted key is recycled and becomes the current key, inserted next.

15 If the insertion location was empty, the thread has finished inserting its key and exits.

18 The key was not inserted. The age is incremented and the next insertion location will be tested.

After construction the maximum ages stored in `MAT` are optionally quantized and packed together with the keys in `H`. This is done in a second CUDA kernel. The table `MAT` is discarded after this. Note that duplicate keys in the input could be trivially handled by comparing whether `prev` equals `age_key`.

Running the kernel The number of threads and groups is chosen so as to maximize the GPU workload. A thread that has finished its job sits idle until all threads in its group have finished as well. While our coherent probe sequence and max-age table help reducing thread divergence, some remain and idle threads do occur. To minimize their number, the number of threads per group should not be too large. It should however not be too small either for a good GPU utilization as we are limited by the maximum number of groups working simultaneously (120 in our NVidia Fermi GPU). We experimentally found that the best tradeoff is to use 192 threads per group.

Thus, we set the number of groups to $\lceil |D|/192 \rceil$ (a few threads after the end of the input array run without performing any operations).

4 Results and discussion

The performance of our scheme is impacted by several factors: The number of keys to be hashed, the target load factor, and whether coherence exists in the data and the access. All our tests are performed with a NVidia Fermi GTX 480 GPU.

In the following comparisons we introduce a variant of our scheme using a random probe sequence h_{rand} defined as follows:

$$h^i_{\text{rand}}(k) = c_0 + (k * c_1) + (i * c_2) \mod |H|$$

where c_0, c_1 and c_2 are large random numbers. The purpose of h_{rand} is to reveal when coherence is successfully exploited by our coherent probe sequence. Indeed, in absence of coherence we expect h_{coh} and h_{rand} to result in similar performance. When coherence is available we expect better performance from h_{coh}.

We compare our results to the methods of Alcantara *et al.* [2009; 2011]. We use the implementations made available by the authors in the NVidia CUDPP library, using the multiplicative hash functions described in these papers. We ran all tests on a NVidia GeForce GTX 480.

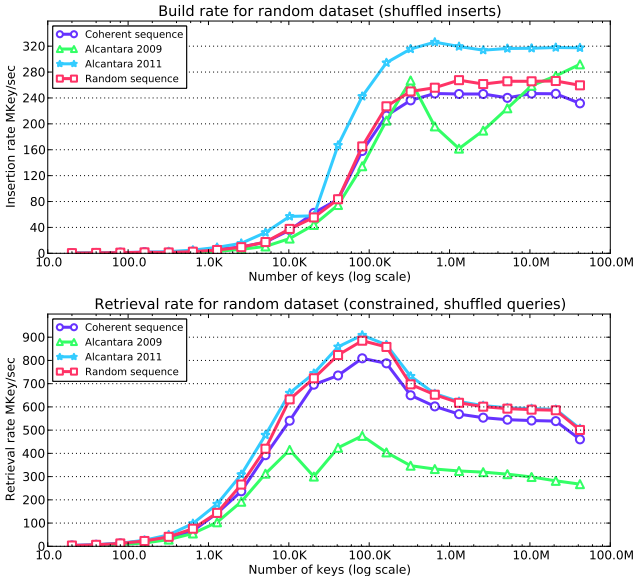


Figure 5: *Insertion rates (top) and retrieval rates (bottom) for h_{rand} and h_{coh} and earlier schemes, for increasingly larger input, under 0.8 load factor. Timings are averaged over several runs. Please refer to the text for details on these data sets.*

In Section 4.1 we analyze the behavior of our hash when no particular coherence exists, and compare it to previous work. In Section 4.2, we discuss the behavior of our hash in a Computer Graphics setting, where spatial coherence exists.

4.1 Hashing generic data

In this section we focus on hashing random keys taken in a 1D universe, assuming that no particular coherence exists neither between keys nor in the access patterns. We consider key–data pairs of 64 bits, having a key on 32 bits, a data record on 28 bits, and using 4 bits to encode the maximum age. We randomly select an increasingly larger number of keys in the universe of 2^{32} possible keys.

We analyze insertion and retrieval of defined keys. For insertion the input is a vector of key–data records. For retrieval, the input is a vector of keys for which data must be retrieved. To avoid all bias in the measure, we shuffle the input vectors for both construction and query. Of course, this setting exhibits no coherence and corresponds, in fact, to the worst case scenario for our hash. We will see in Section 4.2 that performance significantly increases in the presence of coherence.

Insertion Figure 5 (top) compares construction performance of increasingly larger random sets of keys, under a 0.8 load factor. We observe that both probe sequences h_{rand} and h_{coh} behave similarly. This is explained by the fact that the random input data does not exhibit any coherence that could be exploited during construction.

Retrieval Figure 5 (bottom) compares query performance of increasingly larger sets of keys, under a 0.8 load factor. Again, on these random data sets we observe that both probe sequences h_{rand} and h_{coh} behave similarly. In these tests we only query *defined keys*. Since the input is extremely sparse, there is no coherence in the access. We will observe the benefit of coherence in Section 4.2.

Comparison to Cuckoo hashing Under a 0.8 load factor and for 16M (million) keys, our scheme achieves an insertion rate of 249 Mkeys/sec. In comparison, [Alcantara et al. 2011] achieves 318 Mkeys/sec and [Alcantara et al. 2009] achieves 268 Mkeys/sec. Therefore, in absence of coherence our insertion scheme is slower than both versions of the cuckoo scheme. This is essentially due to the update of the max-age table `MAT`, and the emulation of the 64 bits `atomicMax`. We will see in the next section, however, that in presence of coherence our scheme performance increases significantly.

Remarkably, our scheme consistently reaches load factor as high as 0.99. For 32 millions random keys under 0.99 load factor, our insertion rate is 112 MKeys/s and the retrieval rate is 241 MKeys/s.

Failure rates In all our tests our scheme *never* failed to build a hash table both with h_{rand} and h_{coh}, and up to 0.99 load factor. Load factors higher than 0.99 typically generate a max age above 15 which no longer fits our simple 4 bits encoding. This robustness is an important advantage compared to the Cuckoo scheme where restarts can lead to inconsistent performance behavior under high load factors.

Cuckoo hashing [Alcantara et al. 2009] rarely fails at 0.7 load factor, however this behavior quickly degrades at higher load factors. Higher load factor may be reached by relying on additional tables, at the cost of a reduced query performance.

4.2 Hashing in a Computer Graphics setting

Our scheme is best suited when data is coherent—defined keys tend to be neighboring—and when the data is accessed in a coherent manner. Coherence in the data helps the construction process; However a random set of keys still benefits from a coherent access due to thread locality.

In a typical Computer Graphics application the hash table stores a sparse, structured, set of elements (texels, vector primitives, voxels, particles, triangles) which are accessed with some degree of spatial coherence. In most scenarios, a large number of empty keys are also queried.

2D data We first consider 2D data sets. Our test consists in hashing a sparse subset of the pixels of a 2D image (e.g. all the non white pixels), and then query *all* pixels of this image to reconstruct it. In the tests below, keys are computed from the 2D pixel coordinates with a row-major ordering. We later discuss the impact of different pixel orderings.

Figure 6 reports construction times for both a random data set and the fish data set. The important observation is that coherence in the fish data set—the existence of many neighboring keys—directly results in improved construction performance. The fish data set contains $20.5M$ keys. Under a 0.85 load factor, the random sequence reaches 142 Mkeys/s while our coherent hashing scheme achieves 368 Mkeys/s – an improvement of 159%. Thanks to coherence our scheme is now on par with parallel cuckoo hashing for construction times. In contrast, on random data the coherent sequence has similar performance as the random sequence.

Figure 7 reports the time taken to retrieve *all* the keys, both defined and empty. The distinction between querying empty or defined keys is important since empty keys are typically the most expensive to retrieve. In our scheme their cost is greatly reduced by using the the max-age table. The results are shown in Figure 7 for both a random set of keys and the fish dataset. Clearly, both the fish and random data sets benefit from coherence in the access. These results

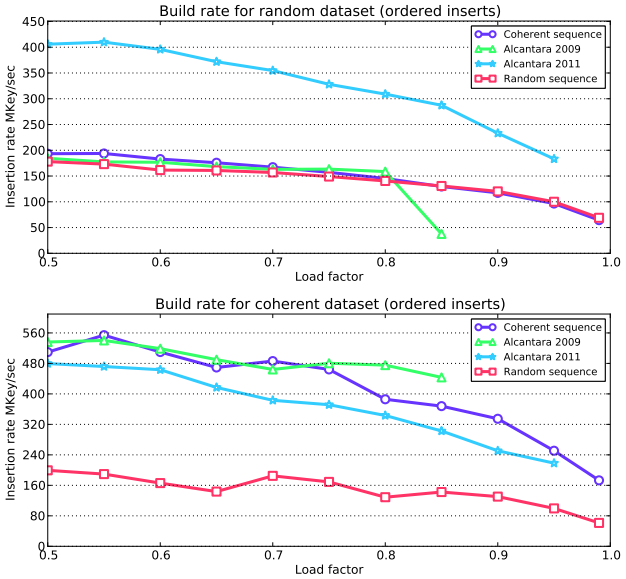


Figure 6: *Construction times for h_{rand} and h_{coh} and earlier schemes. Times are averaged over several runs.* Top: $1M$ *random keys are inserted, taken from a universe of size* $8K \times 8K$. Bottom: *Timings for the* fish *image:* $20.5M$ *are defined in a universe of size* $8K \times 8K$.

are consistent across all the datasets we tested. Under a 0.85 load factor our coherent hash retrieves 5324 Mkeys/s, while all other schemes achieve less than 1000 Mkeys/s: Coherence brings a very significant improvement in query performance.

The benefit of our coherent probe sequence is also clearly revealed by the *percentage of global cache hit* during queries, as shown Figure 8. No other scheme in our tests made any significant use of the cache. Figure 8 shows only L1 cache data. We ran additional tests to reveal further improvements in the number of L2 cache read requests and DRAM read requests, as reported in Figure 9 together with the measured branch divergence rate. In Figures 8 and 9, the data is taken from the above experiment with the fish data set.

Key layout We now analyze the effect of different orderings of the 2D data in the 1D key universe. This is important as in many graphics applications of hashing, the keys are queried in a systematic and coherent way. For example, threads in a same group rasterize neighboring pixels that have neighboring texture coordinates, so we should strive to keep this coherence when translating the position or the texture coordinates into 1D keys. We test three orderings:

- The *linear row-major* order maps (x, y) to $x + Wy$ when W is the width of the (rectangular) domain: we should benefit from the coherent hash when we query neighboring keys on a same line of the domain.
- The *Morton* order maps (x, y) to $\mathcal{M}(x, y)$, the integer obtained by interleaving the bits of the binary representation of x and y. This improves locality along both X and Y axes.
- The *bit-reversal* permutation $\sigma = (\sigma_i, i \in [0, 2^w))$ is obtained by reversing the bits of the binary representation $b_1^i b_2^i ... b_w^i$ of integer i: $\sigma_i = b_w^i b_{w-1}^i ... b_1^i$. For the experiment, we map (x, y) to $\mathcal{M}(\sigma_x, \sigma_y)$. This mapping exhibits no coherence at all.

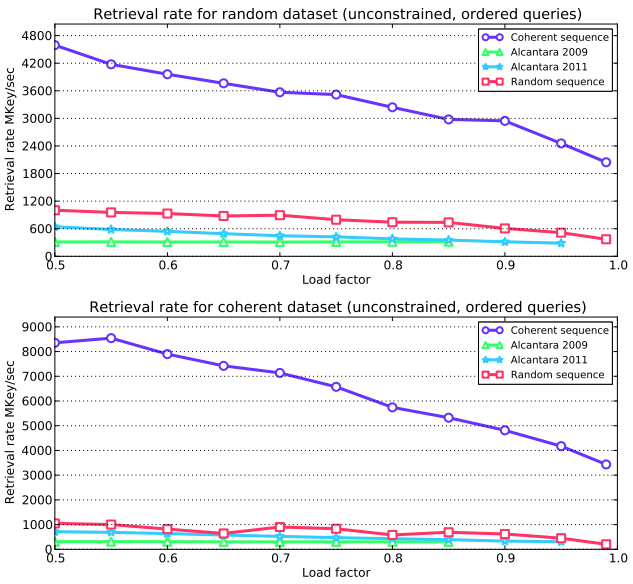


Figure 7: *Access times for h_{rand} and h_{coh} and earlier schemes. Times are averaged over several runs. Missing data for Alcantara09 is due to construction failure at high load factors.* Top: *$8K \times 8K$ keys are queried, $1M$ of which, chosen at random, are defined.* Bottom: *Timings for the fish image: $8K \times 8K$ keys are queried, of which $20.5M$ are defined.*

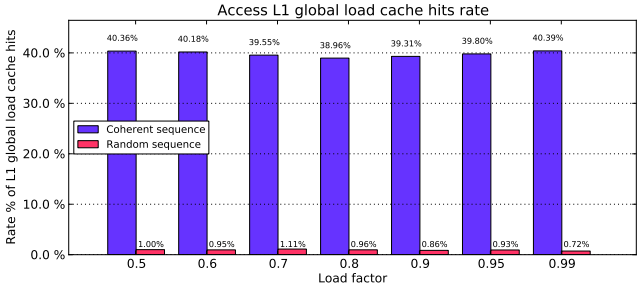


Figure 8: *Percentage of L1 global cache hit during queries for the fish data set. The higher the better. Note that only our coherent probe sequence exhibits a significant cache reuse.*

The test consists in drawing a full-screen rotating image using a custom GLSL pixel shader to query the hash map. The latter is stored as a 2D texture and encodes the image color data using the three orderings above. Since GLSL offers fewer opportunities for optimization, the test reports overall lower performance than the CUDA implementation. The results are shown in Figure 10. The random probe sequence behaves roughly the same for each ordering and even gives slightly faster queries with the highly incoherent bit-reversal ordering. On the contrary, the coherent probe sequence gives significantly faster queries on the two other orderings since it leverages coherence in the access pattern. One can clearly see how increasingly coherent orderings translate into higher performance.

3D data We have experimented with 3D data as well, consisting in voxelizations of the armadillo and hairy models (see Figure 12) in a grid of size 512^3. We hash 64 bits key-data pairs. Our experiments consisted in drawing slices of the volume at random orientations. The armadillo voxel data results in 9.2M keys. Insertion rate is 280 Mkeys/s under load factor $d = 0.8$ and 254 Mkeys/s at $d = 0.99$. Retrieval rate is 1905 Mkeys/s at $d = 0.8$ and 1182 Mkeys/s at $d = 0.99$. The hairy voxel data results in 24M keys. Insertion rate is 455 Mkeys/s at $d = 0.8$ and 182 Mkeys/s at $d = 0.99$. Retrieval rate is 1736 Mkeys/s at $d = 0.8$ and 1208 Mkeys/s at $d = 0.99$.

	Time	L2 requests	DRAM requests	Branch divergence
h_{rand}	97.1 ms	458.6 M	458.6 M	21.8 %
h_{coh}	12.6 ms	44.2 M	42.9 M	10.4 %

Figure 9: *L2 and DRAM read requests for the fish data set under 0.85 load factor.*

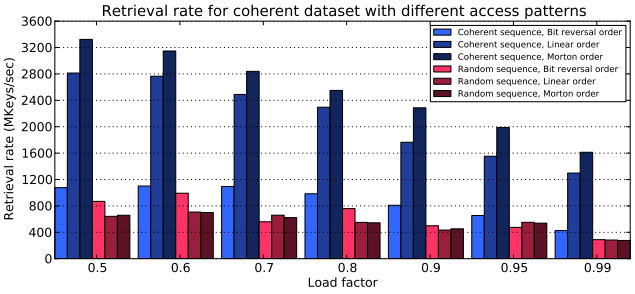


Figure 10: *Query timings for h_{coh} using different ordering of the pixels, for the fish data set.*

Regarding the key layout, we obtain similar results as the 2D results shown in Figure 10, with an even stronger advantage to the Morton ordering.

4.3 Example application

We demonstrate a sparse painting application relying on our hashing scheme, illustrated Figure 11. A 2D atlas is updated interactively while the user paints along the surface. Only the pixels touched by the brush are stored in the hash table. This lets us paint locally at very high resolution, while maintaining a low memory usage.

When the user paints on the model we retrieve the (u, v) coordinates of the pixels touched by the brush. If pixels are already in the hash table, we simply update their colors. If new pixels are touched we rebuild the hash table entirely: We first gather the new pixel key–color pairs and concatenate them with the current hash table from which we remove empty entries. This array is used as the input for building a new hash table. The entire process is fast enough to happen seamlessly while the user paints.

Some applications may choose to spend more memory in exchange for faster queries. This can happen seamlessly by simply rebuilding the hash table with a lower or higher load factor.

For the dataset and viewing conditions of Figure 11, at 0.8 load factor, our scheme with h_{rand} builds in 7 ms and reaches 299 FPS, and our scheme with h_{coh} builds in 3 ms and reaches 375 FPS.

4.4 Limitations, future work

Our scheme has two main drawbacks. First, in absence of coherence in the access patterns our scheme brings little to no benefit compared to a random probe sequence. Second, the max age table slows down construction and requires additional memory. Quantizing the max age could reduce this issue but not solve it entirely. Note that this is only problematic if empty keys are queried: In

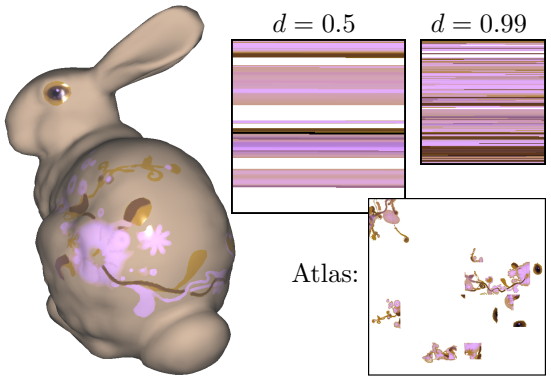


Figure 11: *Our sparse painting application lets the user decorate an object with high resolution details. Only the painted pixels are stored, regardless of the resolution of the virtual texture. In this example the virtual texture has size* 4096^2*, among which* $1M$ *pixels are painted. For this* 1024^2 *viewpoint,* 389586 *queries are made. At* 0.5 *load factor the hash table is built in* 3 *ms and the display runs at* 446 *FPS, while at* 0.99 *load factor it is built in* 10 *ms and display runs at* 237 *FPS.*

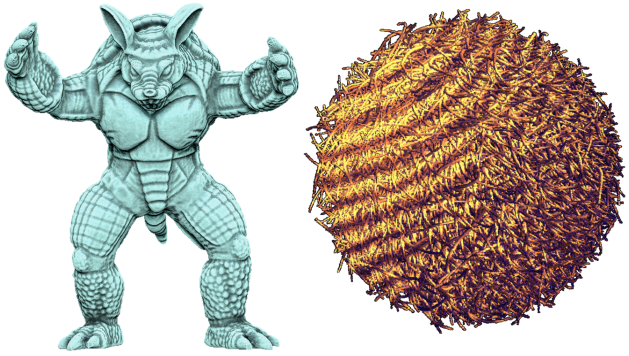


Figure 12: *The armadillo and hairy color voxel data.*

case of constrained access the max age table is not used and does not have to be built.

Future directions of research include handling deletion of keys as well as incremental insertions and deletions while maintaining a compact hash table.

5 Conclusion

Hashing is often synonymous of random access patterns. A remarkable result of our work is to demonstrate that coherence can be preserved, going against this common belief. As shown by our analysis coherence immediately translates to large improvements in cache behavior, and thus to large improvements in query time.

The CUDPP 2.0 implementation of [Alcantara et al. 2011], released concurrently to the publication of our work, also improves significantly the cache behavior. The authors now rely on more coherent hash functions, which resemble the translations of our coherent sequence. This is another strong indication that hashing can preserve memory access efficiency on parallel processors.

Of course, these results only hold when some degree of coherence is available, in the data for construction and in the access patterns for retrieval. This is why we strongly believe our hash is of particular interest for graphics applications, where spatial coherence is common – it was designed from the start with this goal in mind.

In addition, our hash reaches high load factors without failure and performs consistently well at all load factor settings. This is in contrast to Cuckoo hashing which requires to manually select a fixed number of hash functions. Its code is very simple, there is no extra complexity to deal with restarts or to generate new hash functions. We thus believe it offers a strong alternative for storing sparse data in a computer graphics context.

Acknowledgments

We thank Gustavo Patow, Dan Alcantara and our anonymous reviewers for insightful comments and proof–reading. Ms. Sun Qian painted the Stanford bunny. Samuli Laine kindly provided the hairball model; Stanford Computer Graphics Laboratory the armadillo and bunny models; Flickr user Sassy Bella Melange the fish image. Wikimedia Commons user Takkk the flower image. We thank NVidia for donating the Geforce GTX 480 used throughout this work. This work was supported by the Agence Nationale de la Recherche (SIMILAR-CITIES 2008-COORD-021-01), the European Research Council (GOODSHAPE FP7-ERC-StG-205693), and the Ministerio de Ciencia e Innovación, Spain (TIN2010-20590-C02-02).

References

ALCANTARA, D. A., SHARF, A., ABBASINEJAD, F., SENGUPTA, S., MITZENMACHER, M., OWENS, J. D., AND AMENTA, N. 2009. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics (Proc. ACM SIGGRAPH Asia) 28*, 5.

ALCANTARA, D. A., VOLKOV, V., SENGUPTA, S., MITZENMACHER, M., OWENS, J. D., AND AMENTA, N. 2011. Building an efficient hash table on the GPU. In *GPU Computing Gems*, W.-m. W. Hwu, Ed., vol. 2. Morgan Kaufmann, ch. 1.

BOTELHO, F. C., AND ZIVIANI, N. 2007. External perfect hashing for very large key sets. In *Proceedings of the sixteenth ACM Conference on Conference on Information and Knowledge Management*, ACM, CIKM '07, 653–662.

CELIS, P. 1986. *Robin Hood Hashing*. PhD thesis, University of Waterloo, Ontario, Canada.

DEVROYE, L., MORIN, P., AND VIOLA, A. 2004. On worst-case robin hood hashing. *SIAM Journal on Computing 33*, 923–936.

LEFEBVRE, S., AND HOPPE, H. 2006. Perfect spatial hashing. *ACM Transactions on Graphics (Proc. ACM SIGGRAPH) 25*, 3, 579–588.

LINIAL, N., AND SASSON, O. 1996. Non-expansive hashing. In *Proceedings of the twenty-eighth annual ACM Symposium on Theory of Computing*, ACM, STOC '96, 509–518.

PAGH, R., AND RODLER, F. F. 2004. Cuckoo hashing. *Journal of Algorithms 51*, 2, 122–144.

PETERSON, W. W. 1957. Addressing for random-access storage. *IBM Journal of Research and Development 1*, 2, 130–146.